ShnfuCarver Documentation

Release 1.0

Little Tiger

March 15, 2012

CONTENTS

1 Coding Standards **3**
- 1.1 PHP Coding Standards . . . 3
- 1.2 ShnfuCarver Coding Standards . . . 11

2 Development Environment and Tools **13**
- 2.1 Development Environment . . . 13
- 2.2 Development Tools . . . 13

3 ShnfuCarver Design **15**
- 3.1 Framework Architecture . . . 15
- 3.2 Kernel Architecture . . . 16
- 3.3 Framework Components . . . 17
- 3.4 Application Hierarchy . . . 20

4 Shnfu Design **23**

5 About Documentation **25**
- 5.1 Documentation formatting guide . . . 25

6 Indices and Tables **31**

Index **33**

This is the design documentation of `ShnfuCarver` framework, the guide for the whole framework development.

Developers should read this before coding. There is also a pdf version . This documentation could be revised if there is any good suggestion.

CHAPTER

ONE

CODING STANDARDS

1.1 PHP Coding Standards

Developers should always keep in mind all items listed in this chapter. These are the basic conventions for all the PHP coding.

1.1.1 Quick Style Overview

This is an example which includes some of the standards in this chapter. It does not include everything but developers could have a straightforward impression of what will be described in detail in the following sections.

TODO: Write a full example for these coding standards.

```
<?php
?>
```

1.1.2 File Format

Use UTF-8 character encoding for the codes.

Lines must end only with a line feed (LF). Do not use carriage returns (CR) like Macintosh OS do or the carriage return/line feed combination (CRLF) like Windows OS does.

There should be one line feed after the closing PHP tag (?>) at the end of file. Vim would do this automatically and there is no need for adding an extra line.

1.1.3 Coding Style

- No spaces on a blank line.
- No trailing spaces at the end of line.
- No extra blank line at the end of file.
- Add proper blank line to separate different logic.
- Break a long line into several short lines. It is recommended that lines should be no more than 80 columns. This could increase the readability, as 80 is just the length of lines in old text mode.
- Only English character is allowed in the code, including the comments, except for the localization strings.
- Align assignment and parameters in function call when possible:

```
<?php
$result        = $this->callSomeFunction     ('param1',     'second',         true );
$second_result = $this->callSomeOtherFunction('parameter2', 'third',          false);
$last_result   = $this->callSomeFunction     ('3',          'verrrrrrylong', true );
?>
```

1.1.4 Coding Method

- Always use long tags `<?php ?>` instead of short tags `<? ?>`.
- Perl/shell style comments (#) are discouraged.
- The constructor should mainly include class initialization. Do not put in too much action.
- Do not use PHP code (e.g. `echo` or `print`) to output the html tag.
- Use === instead of == if possible.
- Do not use magic numbers.
- Do not do exact comparison between float numbers.
- Avoid embedded assignment. This could lead to misunderstanding.
- Use single quotes `'` for strings if no variable expansion.
- Put each class in a single file.
- Use namespace for clear naming, especially for library.
- Try not to create global variables, which may enhance the relation of modules and introduce unexpected behavior.
- Do not use parentheses for statements. (`require_once`, `include`, `echo`, `return`...)
- Code must be E_STRICT-compatible. This means that it must not produce any warnings or errors when PHP's error reporting level is set to E_ALL | E_STRICT (or E_ALL in PHP 5.4.x).

1.1.5 Including Files

When unconditionally including a class or library file, use `require_once`. When conditionally including a class file (for example, factory methods), use `include_once`.

Either of these will ensure that class files are included only once. They share the same file list, so a file included by `require_once` will not be included again by `include_once`.

1.1.6 Comment

A good program should be understood even without comments. But proper comments could make the reading easier. You do not need to write comments everywhere in the code if the code is in good style. The comment should appear in the following situation:

1. Some complicated algorithm.
2. Where misunderstanding could happen.
3. Some tricks that others may not notice.
4. Any codes where others may not understand or even you may forget the meaning after a few days.

All comments should be written in English, and should in a clear way describe the commented block of code.

1.1.7 Indentation

4 spaces used for indentation.

Do **NOT** use tab. Editors should be configured to treat tabs as spaces in order to prevent injection of tab characters into the source code.

Indentation using Allman style:

```
<?php
$boolVariable   = true;
$stringVariable = 'cat';

if ($boolVariable)
{
    echo 'Bool value is true';
    if ($stringVariable == 'cat')
    {
        echo 'Such a lovely cat!';
    }
}
?>
```

1.1.8 Control Structures

One space between the keyword and the opening parenthesis.

Always use curly braces in control structures, even if they are technically optional. Having them increases the readability and introduces fewer logical errors when new lines added. Start both the curly braces in a new line (Allman style).

Avoid creating too long control structures. A too long control structure may imply a bad design. It's better that it could fit in a single page, which is easy for human to read.

Long control condition could be split into several lines with the logical operators (&&, ||, etc.) at the beginning of the line. However, the long statement of the control condition is not encouraged. Break it into short ones if possible:

```
<?php
// This is too long
if (($condition1
    || $condition2)
    && $condition3
    && $condition4)
{
    // ...
}

// This looks much better
$is_foo = ($condition1 || $condition2);
$is_bar = ($condition3 && $condition4);
if ($is_foo && $is_bar)
{
    // ...
}
?>
```

Condition

An example:

```
<?php
if ((expr_1) || (expr_2))
{
    // action_1;
}
elseif (!(expr_3) && (expr_4))
{
    // action_2;
}
else
{
    // default_action;
}
?>
```

Switch

There must be a `default` statement dealing with default situation. If it is not necessary, also write it following with an immediate `break`.

```
<?php
switch (condition)
{
    case 1:
        // action1;
        break;

    case 2:
        // action2;
        break;

    default:
        // defaultaction;
        break;
}
?>
```

Ternary Operator ?:

A complicated condition could be enclosed in parentheses. Break down too long lines, put the question mark and colon aligned at the front:

```
<?php
$a = $condition1 ? $foo : $bar;

$b = ($condition2 && $condition3)
    ? $foo : $bar;

$c = ($condition4 && $condition5)
    ? $foo_man_this_is_too_long_what_should_i_do
    : $bar;
?>
```

Since PHP 5.3, it is possible to leave out the middle part of the ternary operator. Expression `expr1 ?: expr3` returns expr1 if expr1 evaluates to TRUE, and expr3 otherwise. So the following is an neat way to do this:

```
<?php
$value1 = '';
$value2 = 'some string';

// $result1 will be 'default'
$result1 = $value1 ? : 'default';
// this is identical to the above
$result1 = $value1 ? $value1 : 'default';

// $result2 will be 'some string'
$result2 = $value2 ? : 'default';
?>
```

1.1.9 Array

Assignments in arrays may be aligned. When splitting array definitions onto several lines, the last value may also have a trailing comma. This is a valid PHP syntax and helps to keep code diffs minimal:

```
<?php
$someArray = array
(
    'foo'  => 'bar',
    'spam' => 'ham',
);
?>
```

1.1.10 Function Definition

Arguments with default values go at the end of the argument list.

Try to make functions return something, at least true or false, so it can be determined whether the function call is successful.

```
<?php
function connection($dns, $persistent = false)
{
    if (is_array($dns))
    {
        $dnsInfo = $dns;
    }
    else
    {
        $dnsInfo = BD::parseDNS($dns);
    }

    if (!($dnsInfo) || !($dnsInfo['phpType']))
    {
        return $this->addError();
    }

    return true;
}
?>
```

1.1.11 Function Calls

Functions should be called without space between function's name and starting bracket. There should be one space between every parameter of a function call. Also one space on either side of the equal sign used to assign the return value to a variable. Split long assignment with the equal sign indented on the following line:

```
<?php
$var = foo($bar, $bar2, $bar3);

$GLOBALS['TSFE']->additionalHeaderData[$this->strApplicationName]
    = $this->xajax->getJavascript(t3lib_extMgm::siteRelPath('nr_xajax'));
?>
```

For a long or nested function calls and arrays:

```
<?php
$this->someObject->subObject->callThisFunctionWithALongName
(
    $this->someOtherFunc
    (
        $this->someEvenOtherFunc
        (
            'Help me!',
            array
            (
                'foo'  => 'bar',
                'spam' => 'eggs',
            ),
            23
        ),
        $this->someStillOtherFunc()
    ),
    $this->wowowowowow(12)
);
?>
```

For a chaining call:

```
<?php
$email->from('foo@example.com')
    ->to('bar@example.com')
    ->subject('A great message')
    ->send();
?>
```

Avoid using recursive function because too many levels of recursion may cause stack overflow. Using iteration instead where recursion needed.

1.1.12 Naming Conventions

This is a table for easy search.

Element	Name Example
Namespace	`FooBar\BarFoo`
Interface	`FooBar`
Class	`FooBar`
Method/Function	`fooBar`
Private/Protected Method	`_fooBar`
Variable	`fooBar`
Constant	`FOO_BAR`
Array Key	`foo_bar`

Use meaningful English word(s) for names, do **NOT** use Pinyin. It would be great if others could understand what the code will do just by the names.

Do not use all uppercase abbreviations. `watchNBAGame` should be `watchNbaGame`

Some common prefix to indicate special meaning for methods:

- The prefix `is` should be used for methods returning a boolean value.
- The prefix `set` and `get` for setting and retrieving properties. Do not set properties public.

Prefix for Type of Methods and Properties

Private and protected class members are preceded by a single underscore _.

Do **NOT** use double underscore "__" because it is the prefix of magical constants of PHP.

Interface Name

Class and interface name should be written in CamelCase:

```
<?php
interface Template
{
    public function createProfile($fileName);
}
?>
```

Class Name

Class name should be written in CamelCase:

```
<?php
class UserKey
{
    // ...
}
?>
```

Namespace Name

Namespace should be written in CamelCase:

```
<?php
namespace ShnfuCarver\CoreControl;
?>
```

Method and Function Name

Method name should be like an action and written in camelCase:

```
translate
loadClass
```

Property, Variable and Parameter Name

Written in camelCase:

```
house
someProperty
globalVar
```

Do not use old fashion `var` to declare properties.

Try to initialize the properties in order to set a fixed type:

```php
<?php
class FooBar
{
    private $simpleBoolean = false;
    private $simpleInt     = 0;
    private $simpleFloat   = 0.0;
    private $simpleString  = '';
    private $simpleArray   = array();
    private $simpleNull    = null;
}
?>
```

Constant Name

Constants should be all uppercase, with underscores to separate words:

```php
<?php
class FooBar
{
    const A_CONSTANT_VARIABLE = 1234;
    const ANOTHER_CONSTANT_STRING = 'Haha';
}
?>
```

Constants must be defined as class members with the `const` modifier. Defining constants in the global scope with the `define` function.

Array Key Name

No "–" for magic quote.

```php
<?php
$someArray = array('foo_bar' => 'Pretty good!', 'monday' => 1);
?>
```

1.1.13 Coding Standard Tool

PHP_CodeSniffer.

1.1.14 Reference

This chapter is inspired by the following.

- PEAR Coding Standards
- Zend Framework Coding Standard for PHP
- CakePHP Coding Standards
- PHP Coding Standard

1.2 ShnfuCarver Coding Standards

1.2.1 Database Naming Conventions

Since the framework may use a third party ORM library. It may have its own naming conventions required by the library.

But in general, the name should comply with the following:

- Use only singular form in all the names.
- All names should be lowercase with "_" separating words.
- A table should have a primary key named as "id".
- A foreign key name should be the related table name followed by "_id".

TODO: Write a simple example.

1.2.2 URI Design

All URI should be independent of programming language and file names.

That means the server could be totally rewritten by another programming language without changing any of the URIs.

Generate absolute URI in the pages, not relative URI. In this case a page saved to other place will looks exactly like the original one.

CHAPTER

TWO

DEVELOPMENT ENVIRONMENT AND TOOLS

2.1 Development Environment

2.1.1 Programming Language

PHP

2.1.2 Database

MySQL

2.1.3 Server

Debian

2.1.4 HTTP Server

Nginx and Apache HTTP Server (with php module). Install these packages via `apt-get`.

Direct access via nginx for static objects (image, js, favicon.ico, robots.txt...)

Rewrite all other URI request to index.php. Proxy php request to apache.

Settings for security. Hide the version information. Also remember to hide the PHP info in the header.

2.2 Development Tools

This chapter describes some utilities, which could assist the development process. Some of them could be picked as you like (e.g. editors), while some are required (e.g. `Git`).

2.2.1 Editor

The vim or emacs is recommended as the program editor.

Vim

See the official site.

Type `:help` in vim to see the vim user manual. A Chinese version of vim user manual is also available.

Emacs

See the Emacs.

2.2.2 Revision Control

Git

The Git Reference is strongly recommended if you are new to Git.

And the more detailed book Pro Git is good for more comprehensive knowledge. It also has a Simplified Chinese Version of Pro Git.

CHAPTER

THREE

SHNFUCARVER DESIGN

Note: Need a proper name for this framework.

3.1 Framework Architecture

`ShnfuCarver` is designed to be the framework of Shnfu. It should be powerful and efficient for the future `Shnfu` application design.

`ShnfuCarver` is subject to the new BSD license. The license file `license.txt` must be placed under the root directory of the distributed files.

3.1.1 Model View Controller (MVC) Pattern

`ShnfuCarver` uses the MVC pattern. A detailed article about MVC could be found in "Model View Controller (MVC) in PHP".

MVC is widely used among a great many PHP frameworks. Like Symfony, CakePHP, CodeIgniter, Zend Framework, etc.

Use of the MVC pattern results in separating the different aspects of the application (business logic, and UI logic, input logic), while providing a loose coupling between these elements. This could make the code more flexible and maintainable.

3.1.2 Front Controller Pattern

`ShnfuCarver` uses the front controller pattern.

`index.php` is often used as the front controller. All dynamic URI requests are passed to and dispatched by `index.php`. The rewrite trick could be used to hide the `index.php` in the URI.

Of course script with other name is also fine. e.g. while the directory index is set to the static page `index.html` then the front controller could be `do.php` or some page else, while the rewrite trick is still powerful for rearranging the URI.

In the front controller, a typical procedure is like the following:

```
<?php
define('APPLICATION_PATH', realpath(__DIR__ . '/..'));
require_once APPLICATION_PATH . '/Application/Application.php';
```

```
$application = new AppManager();
$application->main();
?>
```

3.1.3 Active Record Pattern

Active record is an approach to accessing data in a database.

There is no need to developing a new library. Just test some third party libraries and choose a suitable one. It will not affect the kernel component of the framework.

Here are some choices:

- Doctrine
- Propel
- NotORM

3.2 Kernel Architecture

The various functionalities are provided by `Service`.

The basic component of application is `Manager`.

`Controller` is the process unit for a single request.

3.2.1 Service

The `Service` is the way of invoking various functionalities. Each service contains a name and the real object.

A `ServiceRepository` is responsible for handling the services. One could retrieve a service by its name from the repository.

3.2.2 Manager

The `Manager` is the most important part of the application. Manager could have daughter managers. The top most manager is the `AppManager` which holds all other managers.

All managers are instances of `ManagerInterface`. A manager have the following basic methods:

- getName
- init
- run
- clean
- loadConfig
- loadAutoload
- setServiceRepository

There may be other common methods that should be added.

A manager has its own configuration via the name. Make sure there are no two managers with the same name.

A manager may have its own autoload settings.

All managers share a ServiceRepository instance, where the manager could register and retrieve services. The manager may make a shortcut method for acquiring the services.

3.2.3 Controller

All controllers are instances of `ControllerInterface`.

The controllers also share a ServiceRepository instance, but they should only retrieve services. They are not supposed to register services which is the responsibility of managers. The controller may also make a shortcut method for acquiring the services.

3.3 Framework Components

The ShnfuCarver should have the following components.

- Config
- Debug, Error, Exception
- Loader, Autoloader
- Dispatcher, Router, Request, Response
- View, Layout, Helper
- Model
- Log

3.3.1 Config

In ShnfuCarver, most settings are configurable via `Config` module:

There will be global and local config. An application have a global one while each manager could have their own.

- File and module paths. The only paths which should be set externally are the ShnfuCarver path and the configuration path itself.
- System settings. Error level, log level.
- Load and autoload.
- Data source.
- ...

The `Config` module does not directly interact with any other modules except `Manager`. All settings are retrieved and set to other modules by `Manager` in order to reduce the coupling.

A default configuration is designed internally. So the application could work well with the minimal settings.

An application could have many configurations which can be switched by the configuration name, like “development”, “production”, etc. One configuration could be originated from another one and then modified.

3.3.2 Debug

Error

Using observer pattern for error handler. User can add its own handler. A handler that implements HanderInterface could be appended to be an observer.

The `Error` module should deal with the PHP system and user errors. The system error reporting level can be set by the application. Errors can be displayed in the web page and/or put into the log module.

Exception

Like the `Error` module, using oberver pattern to handle uncaught exception. Implements HandlerInterface to customize your own method.

3.3.3 Loader

The load of class could be set manually by configuration. An automatic mechanism is provided if the class file path follows certain regulation.

A map is provided to connect between names and paths. The path could be an array.

The `Loader` process is shown below while the direction of search could be reversed. This process ends while a file is found and loaded:

1. Search the class name in the file map.
2. If not found, extract the namespace from the class name. Use empty string '' for no namespace. If sub-namespace exists, choose the empty or longest one (depend on the direction of search). There will be one or several directories connected with the namespace. Then search the file with the same name as the class.
3. If not found, try to extend or shorten the name of the sub-namespace with one level. Then do the same as above. If still not found, repeat this step. This step is ended by null namespace.

Here is an example. For the class `\ShnfuCarver\Controller\Error` with reversed search direction.

1. Search the file path `\ShnfuCarver\Controller\Error` in the map. If the file found, just load it and end this search.
2. Search `\ShnfuCarver\Controller` in the directory map. If found, search `Error.php` under the directory or directories.
3. Search `\ShnfuCarver` in the directory map. If found, search `Controller/Error.php` under the directory or directories.
4. Search an empty namespace in the directory map. If found, search `ShnfuCarver/Controller/Error.php` under the directory or directories.

Autoloader

The `Autoloader` will hold many `Loader` instances. The loaders may also be considered as the oberver. Multiple `Autoloader` instance is allowed since the `spl_autoload_register` function could be called repeatedly.

The framework could use the autoload mechanism in development for convenience. But a presetted file map should be built in release version in order to save the cpu time. Maybe a cache mechanism could help here.

3.3.4 Dispatcher

Request

The `Request` includes all the information received from the browser. They may have the Server, Header, Get, Post, File and Cookie modules.

Headers are achieved from the `_SERVER` variable. A lot of other useful information is also analyzed from the server variable.

Server, Get, Post and Cookie data have similar structrue. They could be described together.

Response

The `Response` is the content sent back to the browser. They will have the Header, Cookie and Body modules.

There may be many kinds of headers. A `HeaderCollection` will manager all of them. The status header may be handled separately.

Router

The `Route` module processes the path info and translate it into a command. The command will be used to decide which controller to load. Then the controller should send a response back. A command has properties of controller, action and parameters.

The process of a router is:

```
PathInfo -> Rewrite -> Parser -> Command
```

There will be a standard parser to parse a URI like:

http://example.com/controller.action-param1-param2-param3?option1=value1&option2=value2

The rewrite rule will allow different URI form. It will rewrite the path info first before it send to parser.

A default controller should be set if no controller specified. If no action specified, then the index action will be chosen. If specified controller or action not found, an error controller or action should handle this situation and send a 404 error status. These default and error controller can certainly be customized by application.

Generator

`Generator` is used to generate the URI according to the controller, action and parameters, which is opposite to the `Router`.

The process of a generator is:

```
Command -> Deparser -> Rewrite -> PathInfo
```

3.3.5 View

The `View` module will load a template as the response body.

It is easy to implement a static template. But no parameters could be sent to that.

There might be other choices that could be used as template. Such as Smarty and Twig.

The `View` module is often loaded by a controller's action and send to the response as body.

3.3.6 Model

The `Model` represents the business logic of the application.

The model can use a third party library to communicate with the database.

3.3.7 Log

`Log` may be an instance of `Model`. Writing a log is done by a controller's action.

The `Log` module deals with log information with log level. Which levels written to log could be customized by the application. The log should be located on some permanent storage by the `Model` serialization. A database is a good way to hold the log.

The log style is like the following:

Time	Level	Information	...
2012/03/21 09:23:50	Warning	Unauthorized access to user info	...
2012/04/01 18:07:24	Error	Could not link to database	...

Detailed information of the above table:

- Date and time.
- Log level.
- Log information.
- Module which invoke the log.
- Code file path and line of the log.
- Any more information not included above could be merged into the "Information" column.

Log level includes:

- **Debug**. Some debug information for the developers. This won't appear in a production release.
- **Notice**. Just common information about what happened.
- **Warning**. Something not in right state but could not endanger the system.
- **Error**. Error happened and the system may not work properly.
- **Fatal**. Fatal error happened and not recoverable. The system may be shut down.

3.4 Application Hierarchy

This is a typical application structure tree:

```
ShnfuCarver
|-- Project
|   |-- Public
|   |   `-- ...
|   |-- Application1
|   |   |-- Webroot
|   |   |   `-- index.php
|   |   `-- Application
|   |       |-- Application.php
|   |       |-- Config
|   |       `-- Manager
```

```
|   |           |-- First
|   |           |   |-- Config
|   |           |   |-- Controller
|   |           |   |-- View
|   |           |   `-- Model
|   |           `-- Second
|   |               `-- ...
|   |-- Application2
|   |   `-- ...
|   |-- Application3
|   |   `-- ...
|   `-- ...
|
|-- Library
|   `-- ShnfuCarver
|       `-- ...
|
`-- Plugin
    |-- ThirdParty1
    |   |-- Component
    |   `-- Manager
    `-- ThirdParty2
        |-- Component
        `-- Manager
```

The library is the main directory of the `ShnfuCarver` framework. Plugin exists under root directory. It contains some third party libraries. It could provide some extra functionality to the `ShnfuCarver`.

Different applications can be maintained under the same project. Each application has its own configuration. There might be a `Public` directory which could be shared among all the applications.

Note: This hierarchy is not required but suggested. It can still be customized with the configuration.

CHAPTER

FOUR

SHNFU DESIGN

This page is not supposed to be here. Move it to the proper place when ready.

Maximum the usage of AJAX, i.e. the request page is mostly a static page. All dynamic requests and responses are transfered through AJAX. Try to minimize the number of requests in order to reduce the server's burden.

CHAPTER

FIVE

ABOUT DOCUMENTATION

5.1 Documentation formatting guide

The shnfu documentation is written with ReST formatted text. ReST (Re Structured Text) is a plain text markup syntax similar to markdown, or textile. To maintain consistency it is recommended that when adding to the CakePHP documentation you follow the guidelines here on how to format and structure your text.

5.1.1 Line length

Lines of text should be wrapped at 80 columns. The only exception should be long urls, and code snippets.

5.1.2 Headings and Sections

Section headers are created by underlining the title with punctuation characters at least the length of the text.

- # Is used to denote page titles.
- = Is used for sections in a page.
- − Is used for subsections.
- ~ Is used for sub-subsections
- ^ Is used for sub-sub-subsections.

Headings should not be nested more than 5 levels deep. Headings should be preceded and followed by a blank line.

5.1.3 Paragraphs

Paragraphs are simply blocks of text, with all the lines at the same level of indentation. Paragraphs should be separated by more than one empty line.

5.1.4 Inline markup

- one asterisk: *text* for emphasis (italics),
- two asterisks: **text** for strong emphasis (boldface), and
- backquotes: `text` for code samples.

If asterisks or backquotes appear in running text and could be confused with inline markup delimiters, they have to be escaped with a backslash.

Inline markup has a few restrictions:

- It **may not** be nested.
- Content may not start or end with whitespace: `* text*` is wrong.
- Content must be separated from surrounding text by non-word characters. Use a backslash escaped space to work around that: `onelong\ *bolded*\ word`.

5.1.5 Lists

List markup is very similar to markdown. Unordered lists are indicated by starting a line with a single asterisk and a space. Numbered lists can be created with either numerals, or # for auto numbering:

```
* This is a bullet
* So is this.  But this line
  has two lines.

1. First line
2. Second line

#. Automatic numbering
#. Will save you some time.
```

Indented lists can also be created, by indenting sections and separating them with an empty line:

```
* First line
* Second line

    * Going deeper
    * Whoah

* Back to the first level.
```

Definition lists can be created by doing the following:

```
term
    definition
CakePHP
    An MVC framework for PHP
```

Terms cannot be more than one line, but definitions can be multi-line and all lines should be indented consistently.

5.1.6 Links

There are several kinds of links, each with their own uses.

External links

Links to external documents can be with the following:

```
`External Link <http://example.com>`_
```

The above would generate a link pointing to http://example.com

Links to other pages

:doc:
Other pages in the documentation can be linked to using the `:doc:` role. You can link to the specified document using either an absolute or relative path reference. You should omit the `.rst` extension. For example, if the reference `:doc:'form'` appears in the document `core-helpers/html`, then the link references `core-helpers/form`. If the reference was `:doc:'/core-helpers'`, it would always reference `/core-helpers` regardless of where it was used.

Cross referencing links

:ref:
You can cross reference any arbitrary title in any document using the `:ref:` role. Link label targets must be unique across the entire documentation. When creating labels for class methods, it's best to use `class-method` as the format for your link label.

The most common use of labels is above a title. Example:

```
.. _label-name:

Section heading
---------------

More content here.
```

Elsewhere you could reference the above section using `:ref:'label-name'`. The link's text would be the title that the link preceded. You can also provide custom link text using `:ref:'Link text <label-name>'`.

5.1.7 Describing classes and their contents

The CakePHP documentation uses the phpdomain to provide custom directives for describing PHP objects and constructs. Using these directives and roles is required to give proper indexing and cross referencing features.

5.1.8 Describing classes and constructs

Each directive populates the index, and or the namespace index.

.. php:global:: name
This directive declares a new PHP global variable.

.. php:function:: name(signature)
Defines a new global function outside of a class.

.. php:const:: name
This directive declares a new PHP constant, you can also use it nested inside a class directive to create class constants.

.. php:exception:: name
This directive declares a new Exception in the current namespace. The signature can include constructor arguments.

.. php:class:: name
Describes a class. Methods, attributes, and constants belonging to the class should be inside this directive's body:

```
.. php:class:: MyClass

    Class description

   .. php:method:: method($argument)

   Method description
```

Attributes, methods and constants don't need to be nested. They can also just follow the class declaration:

```
.. php:class:: MyClass

    Text about the class

.. php:method:: methodName()

    Text about the method
```

See Also:

php:method, php:attr, php:const

.. **php:method::** name(signature)
Describe a class method, its arguments, return value, and exceptions:

```
.. php:method:: instanceMethod($one, $two)

    :param string $one: The first parameter.
    :param string $two: The second parameter.
    :returns: An array of stuff.
    :throws: InvalidArgumentException

   This is an instance method.
```

.. **php:staticmethod::** ClassName::methodName(signature)
Describe a static method, its arguments, return value and exceptions, see php:method for options.

.. **php:attr::** name
Describe an property/attribute on a class.

Cross Referencing

The following roles refer to php objects and links are generated if a matching directive is found:

:php:func:
Reference a PHP function.

:php:global:
Reference a global variable whose name has $ prefix.

:php:const:
Reference either a global constant, or a class constant. Class constants should be preceded by the owning class:

```
DateTime has an :php:const:`DateTime::ATOM` constant.
```

:php:class:
Reference a class by name:

```
:php:class:`ClassName`
```

:php:meth:
Reference a method of a class. This role supports both kinds of methods:

```
:php:meth:`DateTime::setDate`
:php:meth:`Classname::staticMethod`
```

:php:attr:
Reference a property on an object:

```
:php:attr:`ClassName::$propertyName`
```

:php:exc:
Reference an exception.

5.1.9 Source code

Literal code blocks are created by ending a paragraph with `::`. The literal block must be indented, and like all paragraphs be separated by single lines:

```
This is a paragraph::

    while ($i--)
    {
        doStuff()
    }

This is regular text again.
```

Literal text is not modified or formatted, save that one level of indentation is removed.

5.1.10 Notes and warnings

There are often times when you want to inform the reader of an important tip, special note or a potential hazard. Admonitions in sphinx are used for just that. There are three kinds of admonitions.

- `..  tip::` Tips are used to document or re-iterate interesting or important information. The content of the directive should be written in complete sentences and include all appropriate punctuation.
- `..  note::` Notes are used to document an especially important piece of information. The content of the directive should be written in complete sentences and include all appropriate punctuation.
- `..  warning::` Warnings are used to document potential stumbling blocks, or information pertaining to security. The content of the directive should be written in complete sentences and include all appropriate punctuation.

All admonitions are made the same:

```
.. note::

    Indented and preceded and followed by a blank line. Just like a paragraph.

This text is not part of the note.
```

Samples

Tip: This is a helpful tid-bit you probably forgot.

Note: You should pay attention here.

Warning: It could be dangerous.

See Also:

Some other information related.

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX

D

doc (role), 27

P

php:attr (directive), 28
php:attr (role), 29
php:class (directive), 27
php:class (role), 28
php:const (directive), 27
php:const (role), 28
php:exc (role), 29
php:exception (directive), 27
php:func (role), 28
php:function (directive), 27
php:global (directive), 27
php:global (role), 28
php:meth (role), 28
php:method (directive), 28
php:staticmethod (directive), 28

R

ref (role), 27